# Dependency Models to Manage Software Architecture

Neeraj Sangal and Frank Waldman
*Lattix, Inc.*

*This article describes a new approach for managing software architectures. It uses inter-module dependencies to specify and manage the architecture of software applications. The technique, based on a matrix representation, is simple, intuitive, and appears to scale far better than the directed graph representations that are used currently. It enables specification and automatic enforcement of architectural intent such as layering and componentization. The article concludes by showing how this approach can be applied to a real application. We build a dependency model to represent the architecture of Ant, a popular Java build utility. We then examine how Ant's architecture has evolved over several versions of the software.*

Software development has a way of becoming difficult over time. While they often start well, software projects begin to bog down as enhancements are made to meet new demands and as development teams change. It takes longer to fix problems because fixes made in one area end up introducing bugs in other areas. In no time, a project becomes so complex that no single engineer understands the whole system. As a result, original design assumptions are lost and the boundary between various parts of the system begins to blur. Systems that started out as modular become monolithic.

A powerful new approach, based on using the Dependency Structure Matrix (DSM), has recently been proposed for specifying software architectures. This approach has been built upon ideas that have actually been around for a very long time. The basic notion of *divide and conquer* has been around since time immemorial. Engineers take a large system and decompose it into subsystems; when a subsystem itself becomes too large, it is in turn split up in a process called *hierarchical decomposition*. The decomposition is done in such a way that closely coupled subsystems are closer to each other, while loosely coupled subsystems are kept farther apart.

The problem in software systems is that it is very easy for developers to create undesirable couplings between subsystems. Often this is done without an understanding of the overall system. As a result, subsystems become tightly coupled over time.

The new approach has two key elements: (1) a precise hierarchical decomposition and (2) explicit control over allowed and disallowed dependencies between the subsystems. DSM is ideal for this approach because it provides a compact representation that can easily scale up to tens of thousands of classes or files, whereas conventional box-and-arrow diagrams become unusable for systems composed of even a few hundred classes.

## Understanding DSM

### *Brief History*

DSM has traditionally been used to model tasks involved in the development of discrete products. The structure of dependencies between tasks helps in understanding which tasks can be done in parallel and which have to be performed sequentially. Cyclic dependencies are indicators of possible rework that may be necessary. Steven Eppinger at Massachusetts Institute of Technology's Sloan School [1] spearheaded the application of DSM within some of the largest companies in the world, including Boeing, General Motors, Intel, and many others. However, applying DSM to software is new.

Software engineers have always understood the importance of dependencies between modules. However, they have tended to visualize the modules and their dependencies as directed graphs, i.e., box-and-arrow diagrams. The Unified Modeling Language (UML) makes extensive use of directed graphs, where a variety of boxes and line types are used to indicate the many types of relationships that might exist between different types of modules. This is useful for detailed design but quickly becomes unwieldy as the size of the application increases.

UML diagrams provide limited value in controlling the actual implementation. Indeed, far from controlling the design of the application, developers feel burdened when they have to do a round trip back to their UML models to keep them synchronized with code. Many development teams simply stop maintaining their UML models beyond the initial stages of the project.

Using matrices in systems engineering has a long history. Systems engineers have used N-squared diagrams to model the input and output flows within their system. Subsequent work by Baldwin and Clark at Harvard Business School [2] employing DSM to model the evolution of the computer industry brought it one step closer to the engineering of software.

The key ideas underlying our approach [3], like most ideas in dependency tools, are not new. The notion of inter-module dependency was articulated by Parnas in his early papers (most notably [4]), and the extraction and exploitation of dependencies has been the subject of many more recent projects. The potential significance of the DSM for software was noted by Sullivan et al. [5] in the context of evaluating design tradeoffs. Similarly, Lopes and Bajracharya [6] have also applied DSM to study the value of aspect-oriented modularization. MacCormack et al. [7] have applied the DSM to analyze the value of modularity in the architectures of Mozilla and Linux.

Our approach, however, is the first application of DSM for the explicit management of inter-module dependencies. More information about DSM and available tools, including those from Lattix, can be found at <www.dsmweb.org>.

Figure 1 A-B: *A Simple DSM Before and After Partitioning*

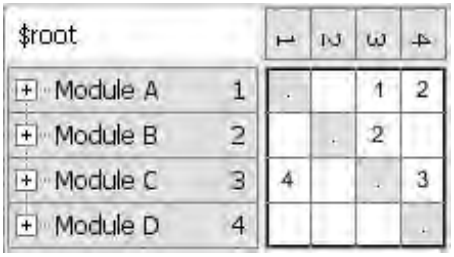



Figure 1A

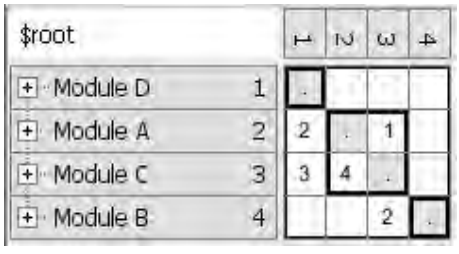



Figure 1B

### *Making Sense of DSM*

Figure 1A shows a simple DSM for a system that consists of four subsystems labeled Modules A, B, C, and D. In the square matrix, the row and column number represent the same module (for compactness, only the rows are labeled). The cells in the grid show the strengths of the interdependencies between each module.

The way to read a DSM is to read the dependencies down a column. For instance, column 1 shows that Module A depends on Module C with dependency strength of 4. Correspondingly, reading across row 1 tells us that Module A provides to Module C and Module D with dependency strengths of 1 and 2 respectively.

Traditionally, DSM has often used an X to indicate a dependency. Also note that a large part of DSM literature reverses our convention of how rows and columns are used. They use rows to indicate *dependencies* and columns to indicate *provides*. However, we have found that our convention is a little more intuitive for software engineers and is also consistent with N-squared diagrams.

Figure 1B shows the DSM after partitioning. Partitioning is a special operation that reorders and regroups modules. The modules are ordered in such a way that those modules that *provide* to other modules are placed at the bottom of the DSM, while modules that *depend* on other modules are placed at the top. If there were no dependency cycles, this would yield a *lower triangular* matrix, i.e., one without any dependencies above the diagonal.

Partitioning also groups together those systems that have dependency cycles. In this case, Modules A and C depend on each other and therefore have been grouped together. This form of the matrix is called *block triangular* because it has been split up into three blocks in which there are no dependencies above the diagonal. Layered systems are naturally expressed as lower triangular matrices.

The grouping of modules can also be shown in different ways. A new compound module can be formed by merging Modules A and C as shown in Figure 2A, after which the matrix becomes *lower triangular*. Notice also that Module D now depends upon the new Module A-C with dependency strength of 5, which is an aggregation of Module D's dependency on both Module A and Module C.

Furthermore, the identities of the basic modules can still be retained by introducing a hierarchy, as in Figure 2B, in which the grouping of A and C is shown by their indentation. The hierarchical

Figure 2A

| $root | 1 | 2 | 3 |
|---|---|---|---|
| Module D 1 | . | | |
| Module A-C 2 | 5 | . | |
| Module B 3 | | 2 | . |

Figure 2B

| $root | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Module D 1 | . | | | |
| A-C: Module A 2 | 2 | . | 1 | |
| A-C: Module C 3 | 3 | 4 | . | |
| Module B 4 | | | 2 | . |

Figure 2 A-B: *The Regrouped DSM and Its Hierarchical Expansion*

decomposition shows that the system has been decomposed into three subsystems: Module D, Module A-C, and Module B. Module A-C is in turn decomposed into Module A and Module C.

This might seem like a simple example but hierarchy is key to scaling with DSM. Hierarchy enables DSM to conveniently model systems with thousands of classes. Hierarchy is also important to the succinct definition of design rules that are used to specify allowed and disallowed dependencies. Design rules can be used to specify architectural patterns such as layering, componentization, external library usage, and other dependency patterns between subsystems. When DSM is combined with design rules, they are called dependency models.

### *Defining a Dependence Relationship*

Before DSM can be applied to software, we need to define the meaning of the statement "a module is dependent upon another module." DSM is a general device that leaves the choice of the definition of dependency to its user. In this case, we are interested in the architecture of the software from *a developer's perspective*. This perspective is important if we are to keep the design modular and to prevent the complexity from spiraling out of control over time.

Therefore, we say that Module A depends upon Module B if the developer of Module A needs to know about the behavior of Module B. The good thing about this definition for software engineering is that, except in a few cases, this can generally be deduced by automatic analysis directly from source code of languages such as Ada, C/C++, and Java. The automatic analysis can be accomplished by utilizing commercially available programs to extract the dependencies. For newer languages such as Java and C#, the dependencies can even be extracted from byte code.

For example, in Java we define a Class A as being dependent on Class B if the following occurs:

1. Class A inherits from Class B (implements in the case of an interface).
2. Class A calls a method or a constructor in Class B.
3. Class A refers to a data member in Class B.
4. Class A refers to Class B (e.g., as in an argument in a method).

The dependency strength can be calculated in a variety of ways. One is to look at

Figure 3 A-D: *Architecture Patterns in a DSM*

| $root | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| com.example: application 1 | . | | | | |
| com.example: model 2 | 44 | . | | | |
| com.example: domain 3 | 33 | 55 | . | | |
| com.example: framework 4 | 100 | 56 | 57 | . | |
| com.example: util 5 | 13 | 12 | 39 | 20 | . |

3A: Layered Pattern

| $root | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| com.example: application 1 | . | | | | |
| com.example: model 2 | 44 | . | | | |
| com.example: domain 3 | | 55 | . | | |
| com.example: framework 4 | | | 57 | . | |
| com.example: util 5 | | | | 20 | . |

3B: Strictly Layered Pattern

| $root | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| com.example: application 1 | . | | | | 1 |
| com.example: model 2 | 44 | . | | | 1 |
| com.example: domain 3 | 33 | 55 | . | | |
| com.example: framework 4 | 100 | 56 | 57 | . | |
| com.example: util 5 | 13 | 12 | 39 | 20 | . |

3C: Imperfectly Layered Pattern

| $root | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| com.example: application 1 | | | | | | | | | | |
| com.example: model 2 | 44 | . | | | | | | | | |
| com.example: domain: tools 3 | 5 | 5 | . | | | | | | | |
| com.example: domain: project 4 | 11 | 16 | | . | | | | | | |
| com.example: domain: comp-1 5 | | | | | . | | | | | |
| com.example: domain: comp-2 6 | | | | | | . | | | | |
| com.example: domain: comp-3 7 | | | | | | | . | | | |
| com.example: domain: services 8 | 5 | 11 | | 12 | 25 | 10 | 13 | . | | |
| framework 9 | 100 | 56 | | 35 | 3 | 4 | 34 | 9 | . | |
| util 10 | 13 | 12 | 5 | 17 | 6 | 6 | 12 | 14 | 20 | . |

3D: Component Pattern

Figure 4A

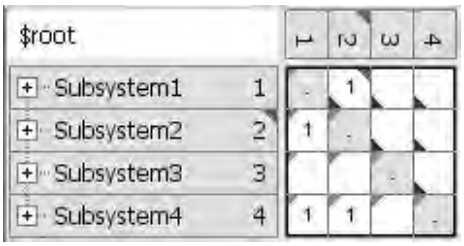



Figure 4B

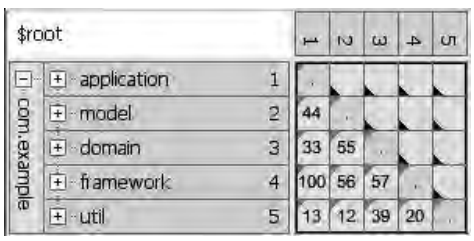



Figure 4 A-B: *DSM with Design Rules*

the number of classes each class depends on; that number is then aggregated in the hierarchy. Yet another possible way is to calculate the dependency strength based on the number of actual uses each class makes of other classes; that number is then aggregated in the hierarchy. Furthermore, it is sometimes useful to filter out specific dependencies such as class references because eliminating them is easy.

## Architecture Patterns in a Dependency Model

### *Layering and Componentization in a DSM*

The DSM representation is uniquely suited for representing certain architectural patterns. Layering is one such pattern. In fact, even when the layering is imperfectly implemented it can still be recognized in a DSM.

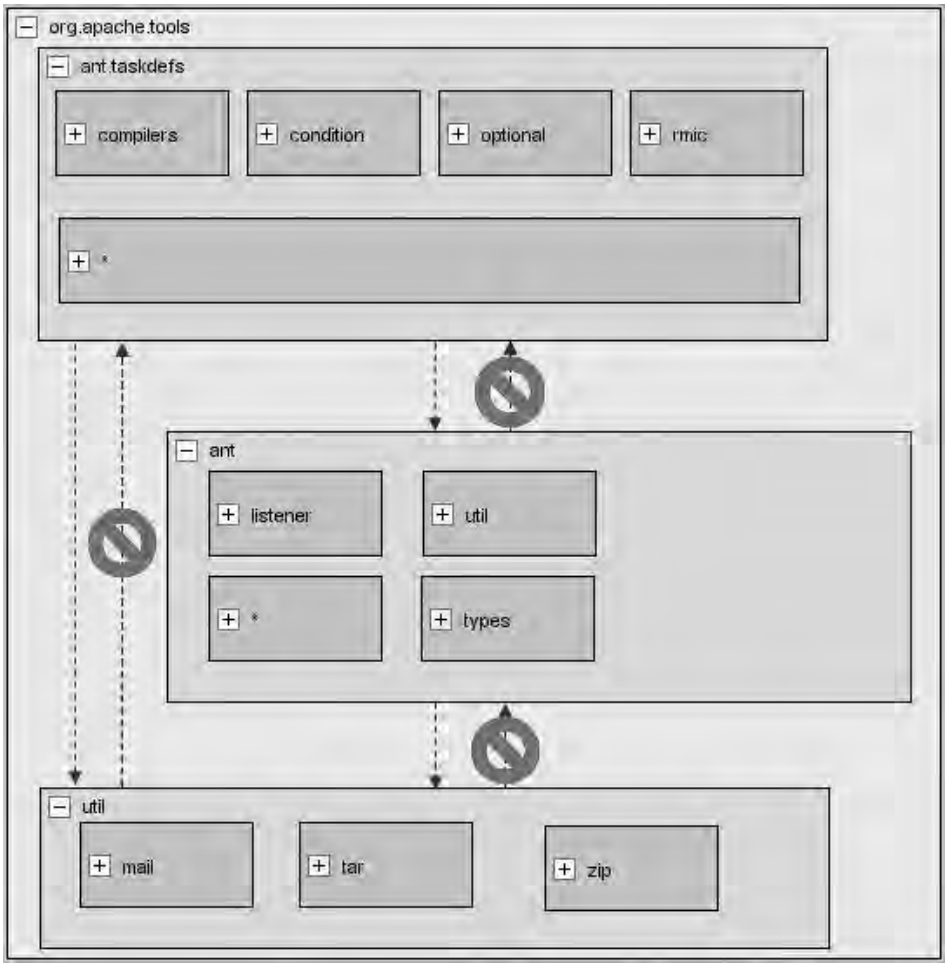
Figure 3A shows the example of a layered system. The figure illustrates that the system consists of five subsystems: *application*, *model*, *domain*, *framework*, and *util*. The DSM shows that the layer at the bottom, util, does not depend on any of the other subsystems; *framework* depends on *util*; *domain* depends on *framework* and *util*; and so on. The lower triangular nature of the matrix makes it immediately apparent that this is a layered system. Figure 3B shows a strictly layered system where each layer depends only on the preceding layer.

Finally, Figure 3C shows an imperfectly layered system. Since the DSM is not lower triangular even after partitioning, we know that there are cyclic dependencies. In this case, the dependencies in column 5 indicate that *util* has dependencies on *application* and *model*. However, the imbalance between the strength of the dependencies suggests that this is an imperfectly layered system.

This discussion must not be construed to suggest that every software application should have a layered design. While a majority of large applications are layered, DSM itself makes no prescription about whether applications should always be layered. However, for those applications that are layered, a representation based on DSM is natural and powerful. As we shall see shortly, design rules allow these architectural patterns to be enforced quite easily.

Figure 3D shows private subsystems *comp-1*, *comp-2*, and *comp-3* within subsystem *domain*. The DSM reveals that nothing in the system depends on these private subsystems. Furthermore, the DSM illustrates that these private subsystems do not depend on each other. This suggests that it is likely that they could be worked upon in parallel once the framework that they depend on is in place.

Figure 5: *Conceptual Architecture of Ant*




### *Design Rules: Enforcing Architectural Patterns*

The architectural patterns that were described in the previous section can be enforced using design rules. The underlying concept of design rules is quite simple: Once you represent the architecture in a hierarchical DSM, then specifying which cells are allowed or disallowed from having dependencies is a natural extension to document and communicate the design intent.

When design intent in the form of design rules is added to a DSM, the result is a dependency model. This dependency model communicates not just what the actual dependencies are but also the allowed and disallowed dependencies. The matrix representation provides a succinct and intuitive visualization for design rules. Figure 4A shows a DSM with design rules expressed as triangles in the corners of the cells. The upper left triangle represents an allowed dependency, while the lower left triangle represents a disallowed dependency and the upper right triangles represent design rule violations. The use of colors can enhance usability; for instance, the triangles can be colored green, yellow, or red to indicate an allowed dependency, a disallowed dependency, or a violation, respectively.

If the DSM grid represents the design space, the design rules qualify that design

space by specifying which parts of the design space are allowed to have dependencies and which are disallowed. In a system with 1,000 classes, a fully expanded DSM grid has one million cells. Since each cell represents design intent, there are one million possible design rules. Fortunately, classes interact with each other in fairly regular ways. Layers are just one example of how classes within each layer interact with classes in other layers. For a five-layer system, just five rules are needed to specify their interaction regardless of the number of classes within the system. Figure 4B shows the design rules for enforcing the layers in such a system. Note that showing only the cannot-use rules tends to make the DSM more readable.

Software degrades from release to release because implicit design rules such as layering are violated. Previously, architects did not have a way to specify, track, or enforce these implicit rules. Dependency models now offer the potential for maintaining the architecture over successive revisions of the life cycle by specifying rules explicitly that define the acceptable and unacceptable dependencies between subsystems. In cases where architecture has evolved and design rules need to be changed, violations can actually make architectural evolution explicit for the entire development team

## Tracking Architecture Evolution in a Dependency Model

We will apply the dependency model approach to Ant, a popular Java build utility. Ant is an open source application that is used by developers to compile, create jar files, build executables, and other sundry development tasks.

One of the principal reasons for the success of Ant has been the notion of tasks. Tasks are named entities that do the actual work and utilize the common Ant framework for doing so. Examples of tasks are *compile*, *copy*, *make directory*, and *create jar file*. Over time, a large number of tasks have been created by independent developers all over the world. This has worked well because the architects of Ant had the foresight to keep the tasks in a layer separate from the framework.

Figure 5 illustrates this conceptual architecture. Ant has been decomposed into three layers: *taskdefs*, *ant*, and *util*. The *taskdefs* layer contains the tasks, the *ant* layer contains the framework, and the *util* layer contains utilities that could be used by both the framework and tasks.

The dotted lines between the subsystems illustrate allowed dependencies, while the dotted lines with a disallow symbol illustrate disallowed dependencies. The layered approach reduces complexity and minimizes the likelihood that a bug introduced in the development of a task will affect the framework, that in turn could affect other tasks.

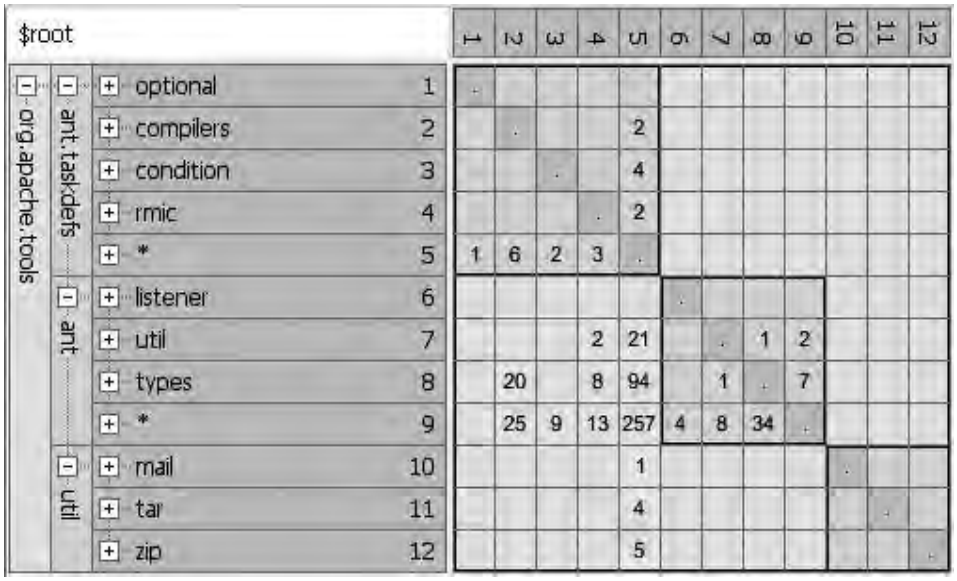


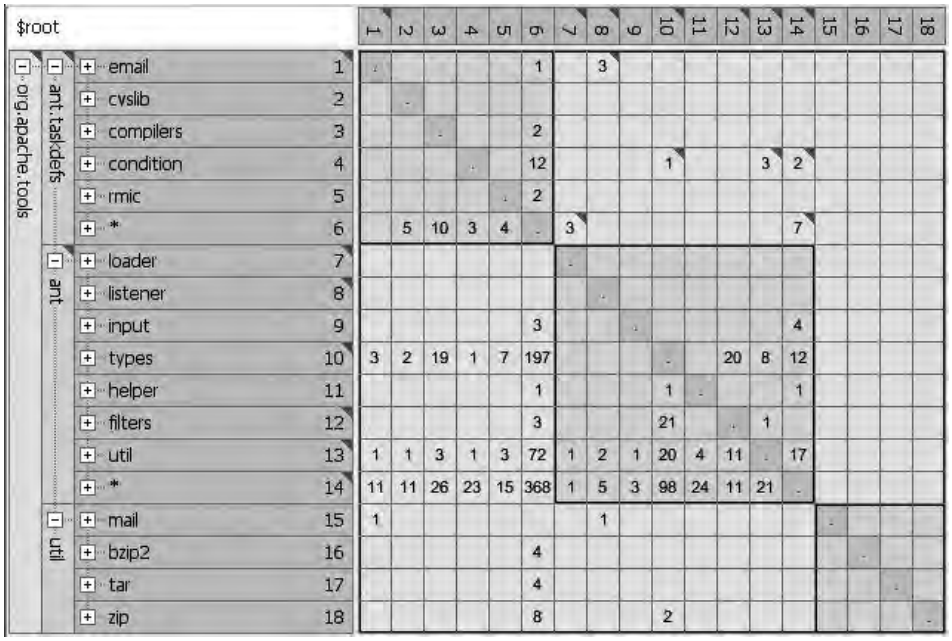

Figure 6: *A Dependency Model for Ant Vers. 1.4.1*

The DSM for Ant Vers. 1.4.1 in Figure 6 shows that Ant has three distinct layers. However, the Ant framework represented by the middle subsystem is largely monolithic. For Vers. 1.4.1, it was not a significant issue because the Ant framework was quite small at the time. We also note that the Java package names do not reflect the layering.

By comparison, the DSM for the current version of Ant in Figure 7 shows that architectural violations have begun to creep in as the Ant framework has become dependent on the taskdefs layer. Further, the application is now considerably larger while the Ant framework continues to be largely monolithic. However, it should be noted that the architecture of Ant is still not as bad as we have seen in many other commercial systems that receive less scrutiny than Ant.

Frequently, systems become so complex that development teams suggest a rewrite. This is inherently a high-risk decision since there is no way to guarantee that the new system will not suffer from similar problems of complexity. The dependency model already gives us guidance about which dependencies to fix and their relative priority. This analysis of the

Figure 7: *Dependency Model for Ant Version 1.6.1*



dependencies reveals where the architecture is really broken and whether remediation requires a complete rewrite or just a fix to the problematic dependencies. Note that additional analysis using the DSM can easily be conducted to explore alternate organization of the architecture, and how the framework itself can be split up so it is no longer monolithic.

Dependency models can also be extended to specify the dependencies of each subsystem upon external libraries. First, the dependency model is examined to see what external libraries are used and which subsystems use them. For instance, in Ant Vers. 1.6.1, we find that some of the external libraries in use include *org.apache.bcel*, *org.apache.bsf*, and *org.apache.xml*. Design rules can then be used to specify the subsystems that are allowed to use these external libraries. Controlling the use of external libraries can pay rich dividends. In addition to helping to maintain architectural integrity, segregating the use of external libraries can also ease the job of migrating to new technologies as they become available.

## Architecture Management as Part of CMMI

The dependency model provides a new way to define and enforce architecture for software engineering process initiatives such as Capability Maturity Model® Integration (CMMI®). A shared understanding of the architecture and managing its change can significantly improve the results that can be derived from these CMMI specific process areas:

- **Requirements Development:** Baseline conceptual architecture and dependency model for inclusion in the technical data package.
- **Technical Solution:** Assess architectural alternatives, formalize precise subsystem decomposition, and define the dependencies between the decomposed subsystems. Measure and verify architectural conformance.
- **Product Integration:** Identify components for product line architectures. Remove redundancies. Improve testability of products.
- **Project Planning:** Estimate impact of new feature requests and track progress against architectural changes.
- **Decision Analysis and Resolution:** Document rationale for architectural changes, understand risks, evaluate, and estimate impact of proposed changes.

® Capability Maturity Model and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

In addition, the dependency model approach provides new means to make architecture management part of the CMMI generic processes. It can be used to manage architecture configurations; involve stakeholders; and to evaluate, monitor, and control the architecture.

## Conclusion

Architecture is integral to software quality. Unless the architecture is explicitly defined, communicated, and controlled, it will degrade as the complexity increases through the life cycle. The dependency model is a powerful new way to manage the architecture of software applications.

It is highly advisable that architecture be tested automatically as part of regular builds. Inadvertent violations can then be fixed immediately, avoiding expensive remediation later in the development cycle. This approach is lightweight – dependency models can be checked and updated automatically with intervention being required only when violations are detected.◆

## References


1. Eppinger, Steven D. "Innovation at the Speed of Information." Harvard Business Review Jan. 2001.
2. Baldwin, C.Y., and K.B. Clark. The Power of Modularity Vol. 1. Cambridge, MA: MIT Press, 2000.
3. Sangal, Neeraj, Ev Jordan, Vineet Sinha, and Daniel Jackson, "Using Dependency Models to Manage Complex Software Architecture." OOPSLA 2005, San Diego, CA. 16-20 Oct. 2005.
4. Parnas, D.L. "Designing Software for Ease of Extension and Contraction." IEEE Transaction on Software Engineering 5.1 (Mar. 1979): 128-138.
5. Sullivan, K., Y. Cai, B. Hallen, and W. Griswold. "The Structure and Value of Modularity in Software Design." Proc. of the 8th European Software Engineering Conference, Vienna, Austria. 10-14 Sept. 2001.
6. Lopes, Cristina Videira, and Sushil Bajracharya. "An Analysis of Modularity in Aspect-Oriented Design." Proc. of Aspect-Oriented Software Development Conference, Chicago, IL. Mar. 2005.
7. MacCormack, Alan, John Rusnak, and Carliss Baldwin. "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code." Harvard Business School. Working Paper No. 05-016.


## About the Authors


**Neeraj Sangal** is president of Lattix Inc., which specializes in software architecture management solutions and services. He has analyzed many large proprietary and open source systems. Previously, Sangal was president of Tendril Software that pioneered model-driven Enterprise Java Beans development and synchronized Unified Modeling Language models for Java. Prior to Tendril, he managed a distributed development organization at Hewlett Packard. Sangal has published and presented papers, most recently a joint work on architecture management presented at the Object-Oriented Programming, Systems, Languages, and Applications 2005 conference.

**Lattix, Inc.**
**8 Harper CIR**
**Andover, MA 01810**
**Phone: (978) 474-5022**
**E-mail: neeraj.sangal@lattix.com**



**Frank Waldman** is vice president at Lattix, Inc. He has extensive experience building companies with innovative technology in a number of industries, including engineering software, consumer electronics, manufacturing, and product development services. Prior to Lattix, Waldman was responsible globally for building markets for the product lifecycle management software business of Eigner, which was acquired by Agile Software to create the largest pure-play product lifecycle management vendor in the global market. He has a Bachelor of Science and Master of Science from Massachusetts Institute of Technology, and holds numerous patents.

**Lattix, Inc.**
**8 Harper CIR**
**Andover, MA 01810**
**Phone: (978) 474-5022**
**E-mail: frank.waldman@lattix.com**